软件分析与验证前沿

孙海英

软件科学与技术系

Big Picture

- Key Concepts
- Taint Analysis Approaches
- Dynamic Taint Analysis Framework
- Typical DTA Applications

Big Picture

- Key Concepts
- Taint Analysis Approaches
- Dynamic Taint Analysis Framework
- Typical DTA Applications

Background

- Taint Analysis is an application of Information Flow Analysis
- Information Flow Analysis enforce a security policy
  - [Confidentiality] secret data does not leak to non-secret places
  - [Integrity] High-integrity data is not influenced by low-integrity data
- Theoretical Foundation of Information Flow Analysis
  - Security Type model based on lattice[1] : **(SC, ⊑)**
  - Non-inference [2]

[1] Denning DE. A lattice model of secure information flow. Communications of the ACM，1976，19(5)：236-243.
[2] Goguen JA，Meseguer J. Security policies and security models. In：Proc. of the'82 IEEE Symp. on Security and Privacy. IEEE. 1982. 1 - 11.

Background

- Taint Analysis
  - 对带污点标记的数据的传播实施分析达到保护数据完整性和保密性的目的[1].
- Theoretical Foundation
  - Model: **(SC, ⊑)** where SC = { Tainted, Untainted }
  - 如果信息从Tainted类型的变量传播给Untainted类型的变量，那么需要Untainted类型的数据改成Tainted类型；
  - 如果Tainted 类型的变量传递到重要数据区域或者信息泄露点，那就意味着信息流策略被违反.

[1] 王蕾, 李丰, 李炼, 冯晓兵. 污点分析技术的原理和实践应用. 软件学报, 2017, 28(4): 860—882

Background

- Taint Analysis Approaches
  - Static Explicit Taint Propagation Analysis
  - Dynamic Explicit Taint Propagation Analysis
  - Static Implicit Taint Propagation Analysis
  - Dynamic Implicit Taint Propagation Analysis

Data Flow based

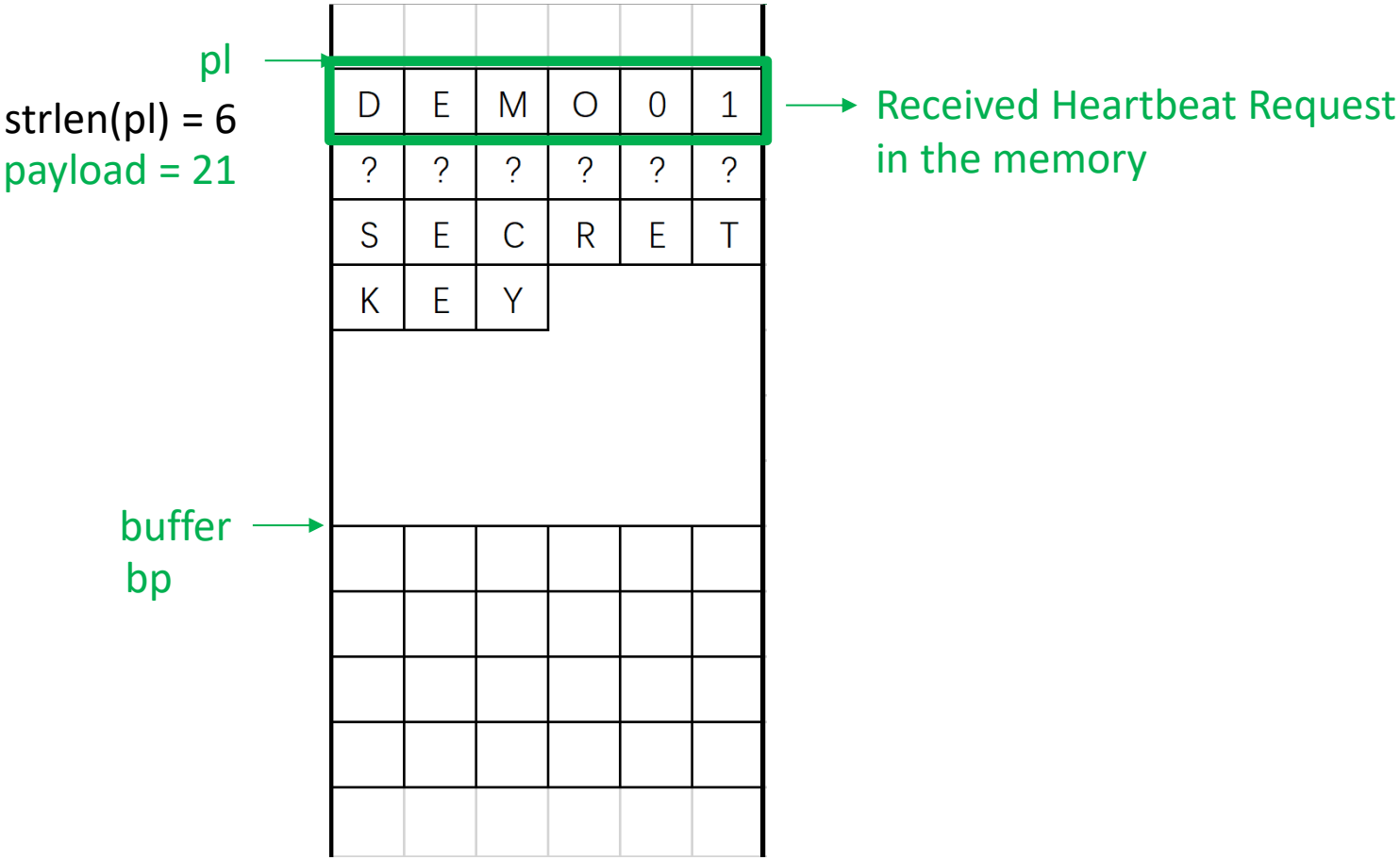
Control Flow based

The Heartbleed Vulnerability[1]

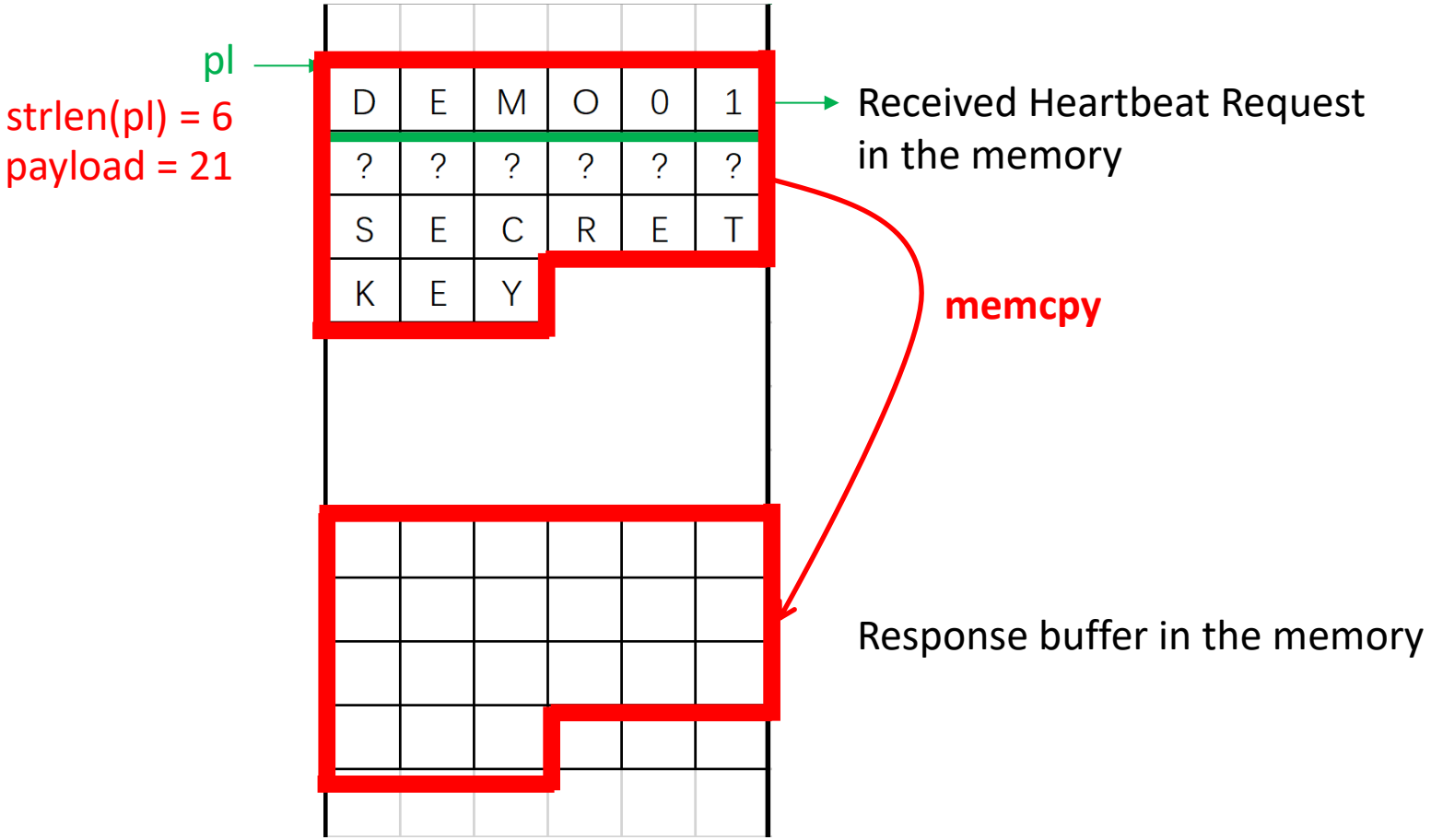
```
2   /* Allocate memory for the response, size is 1 byte
3    * message type, plus 2 bytes payload length, plus
4    * payload, plus padding
5    */
6   buffer = OPENSSL_malloc( 1 + 2 + payload + padding );
7   bp =  buffer;
8
9   *bp++ = TLS1_HB_RESPONSE;
10  s2n( payload, bp );
11
12  /* pl: the point of the payload string
13   * payload: the length of the payload string
14   * pl and payload come from request
15   */
16  memcyp( bp, pl, payload );
17  bp += payload;
18
19  RAND_pseudo_bytes( bp,padding );
20  r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

[1] 丹尼斯 安德里斯著, 刘杰宏等译. 二进制分析实战. 人民邮电出版社，2021.

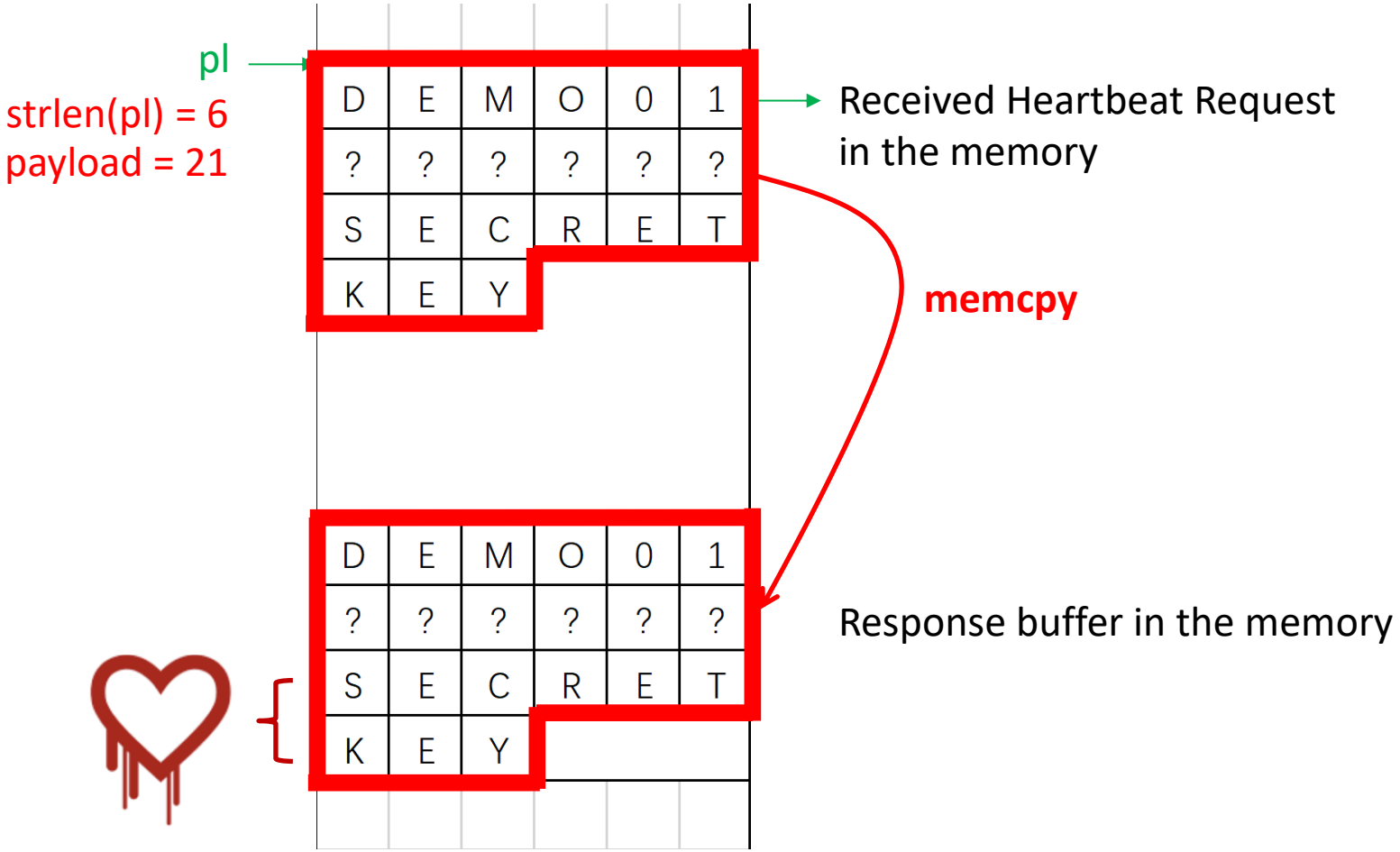
The Heartbleed Vulnerability



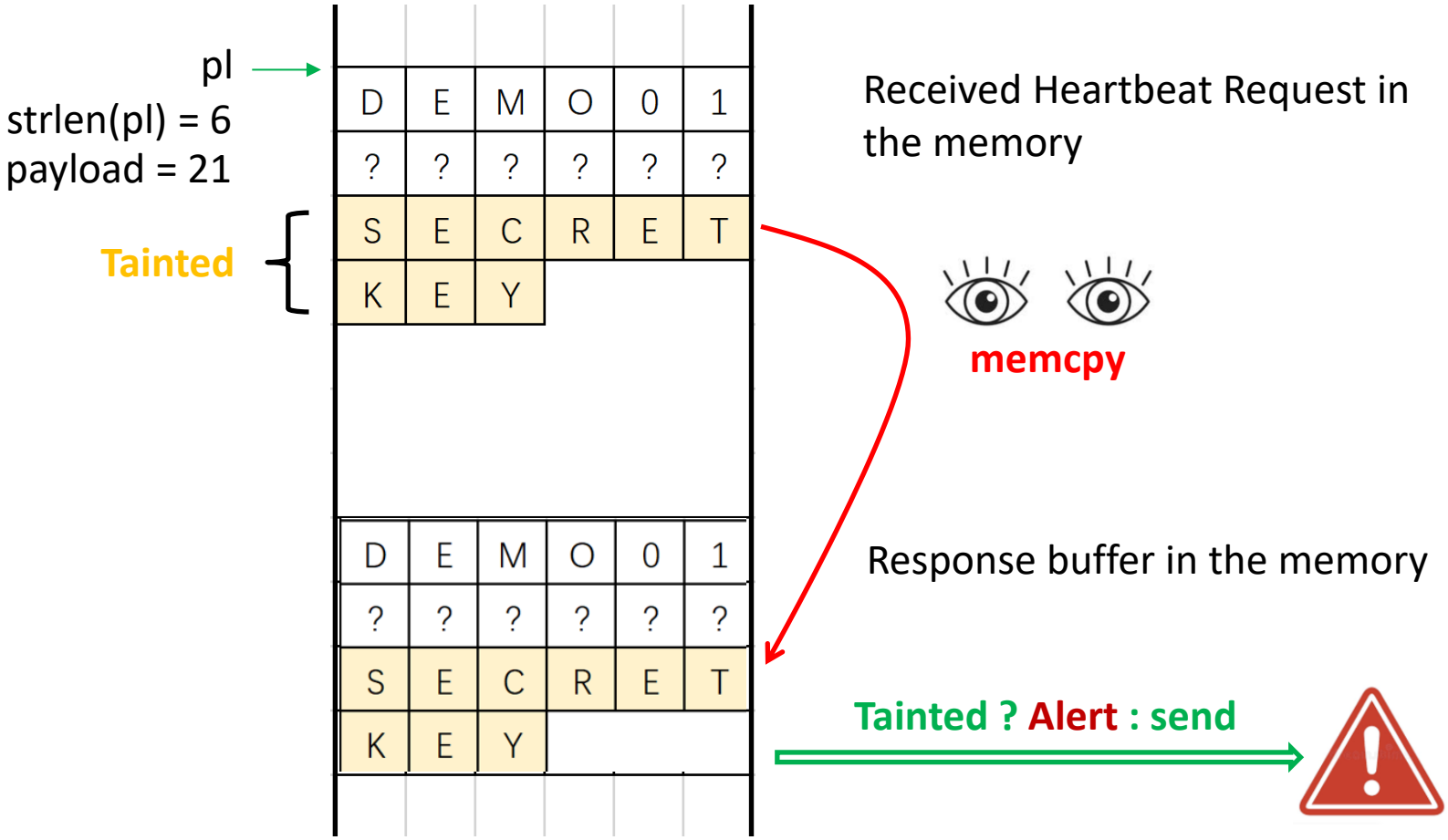
The Heartbleed Vulnerability



The Heartbleed Vulnerability



The Taint Analysis Solution



Key Concepts

- Taint Source (污点源)
  - 所选择追踪的数据所在的程序位置：系统调用、函数入口点或指令
  - 代表着引入不信任数据或机密数据进入系统
- Taint Sink (污点槽)
  - 进行检查的程序位置，确定这些位置是否会收到污点数据的影响
  - 代表直接产生安全敏感操作(违反数据完整性)或者泄露隐私数据到外界(违反数据保密性)；
- Taint Propagation Analysis(污点传播分析)
  - 分析程序中由污点源引入的数据是否能够不经无害处理而直接传播到污点槽
  - 污点传播策略：定义了输入与输出操作之间的污染关系

Exercise

```
11      //... extra code
12      String userName = usernameText.toString();
13      String pwd = passwordText.toString();
14      String smsPwd = "leaked" + pwd;
15      String smsMessage = "User: " + userName + "| Pwd: " + smsPwd;
16      SmsManager smsManager = SmsManager.getDefault();
17      smsManager.sendTextMessage("+86 1234",null,smsMessage,null,null);
18      String sanitizedPwd = Encrypt(smsPwd);
19      String encryptSmsMessage = "| Pwd: " + sanitizedPwd;
20      //... extra code
21  }
```

Taint Source

- Question

1. Where should be set as taint sink?
2. How does the taint data be propagated?

Exercise

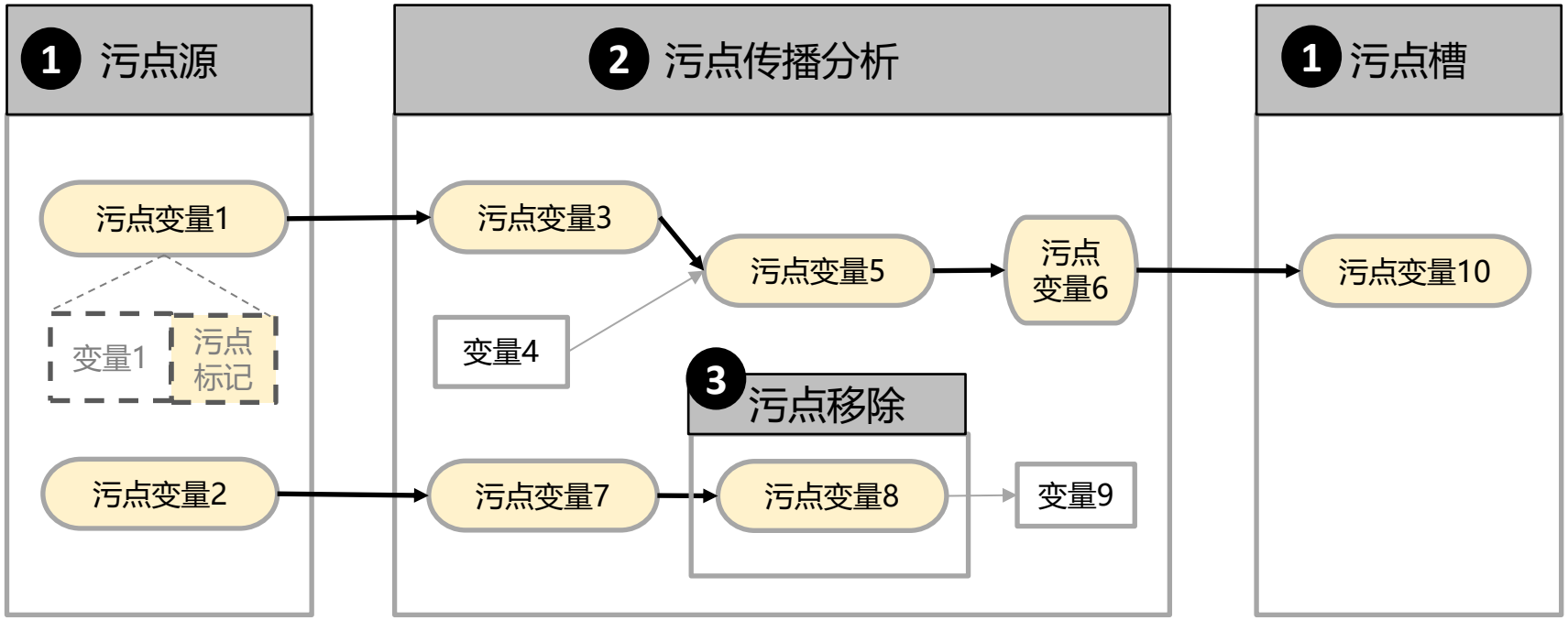
- Question

1. Where should be set as taint sink?
2. How does the tainted data be propagated?

Big Picture

- Key Concepts
- Taint Analysis Approaches
  - General Process
  - Static and Dynamic Analysis Approaches
- Dynamic Taint Analysis Framework
- Typical DTA Applications

The General Process



1 识别污点源和污点槽

2 污点传播分析

3 污点移除

The General Process

- Step1: Identify Taint Source and sink
  - [No general techniques] different system models/program languages/taint objectives…
  - [Who] User (Manual) /Taint Tools (Automatic)
  - [How]

1. 使用启发式的策略：把来自程序外部输入的数据统标为“污点”数据，保守地认为这些数据有可能包含恶意的攻击数据
2. 根据具体应用程序调用的API或者重要的数据类型，手工标记源和汇聚点
3. 使用统计或机器学习技术自动地识别和标记污点源及汇聚点

The General Process

- Step 2: Taint Propagation Analysis
  - 4 Approaches

1. Static Explicit Taint Propagation Analysis
2. Dynamic Explicit Taint Propagation Analysis

Data Flow based

3. Static Implicit Taint Propagation Analysis
4. Dynamic Implicit Taint Propagation Analysis

Control Flow based

The General Process

- Step 2: Taint Propagation Analysis
  - 4 Approaches
  1. Static Explicit Taint Propagation Analysis
  2. Dynamic Explicit Taint Propagation Analysis
  3. Static Implicit Taint Propagation Analysis
  4. Dynamic Implicit Taint Propagation Analysis

Data Flow based

Control Flow based

Static Explicit Taint Propagation Analysis

- [说明] 在不运行且不修改代码的前提下，通过分析程序变量间的数据依赖关系来检测数据能否从污点源传播到污点槽
- [对象] 程序的源码或中间表示.
- [技术] 将对污点传播中显式流的静态分析问题转化为对程序中静态数据依赖的分析
  1. 根据程序中的函数调用关系构建调用图(call graph)
  2. 在函数内或者函数间根据不同的程序特性进行具体的数据流传播分析：直接赋值传播、通过函数(过程)调用传播、通过别名(指针)传播.

Static Explicit Taint Analysis Example

```
26      void demoMethod(){
27          DataClass a  = new DataClass();
28          int b = source();
29          int c = b + 10;
30          foo(a,a,c);
31      }
32  @   void foo(DataClass x, DataClass y, int z){
33          x.a1 = z;
34          sink(y.a1);
35      }
```

Leaked !

A Java Code Example

- Question

1. If the code written in C/C++, does the tainted data still be leaked?

The General Process

- Step 2: Taint Propagation Analysis
  - 4 Approaches
    1. Static Explicit Taint Propagation Analysis
    2. Dynamic Explicit Taint Propagation Analysis
    3. Static Implicit Taint Propagation Analysis
    4. Dynamic Implicit Taint Propagation Analysis

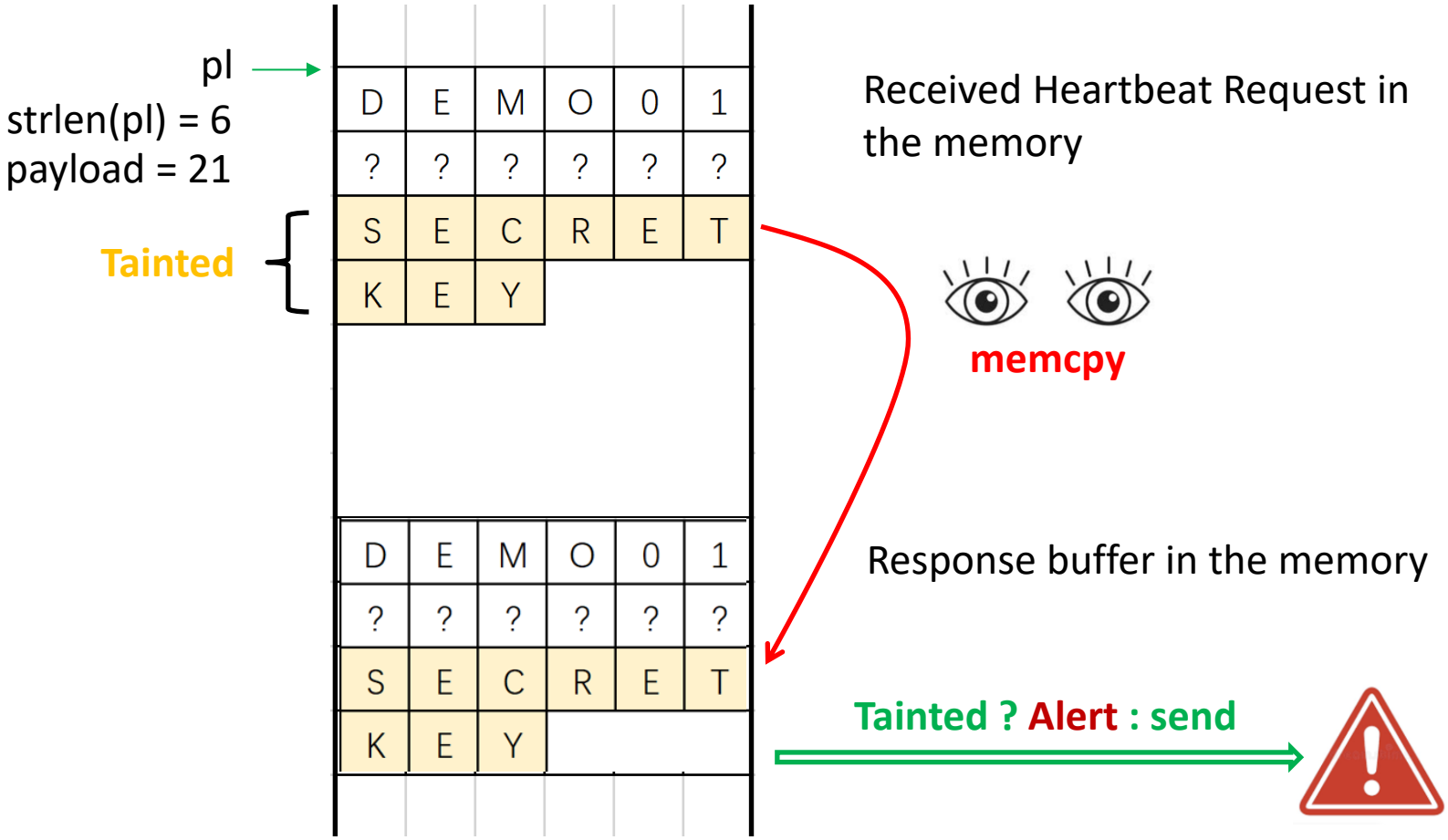
Data Flow based

Control Flow based

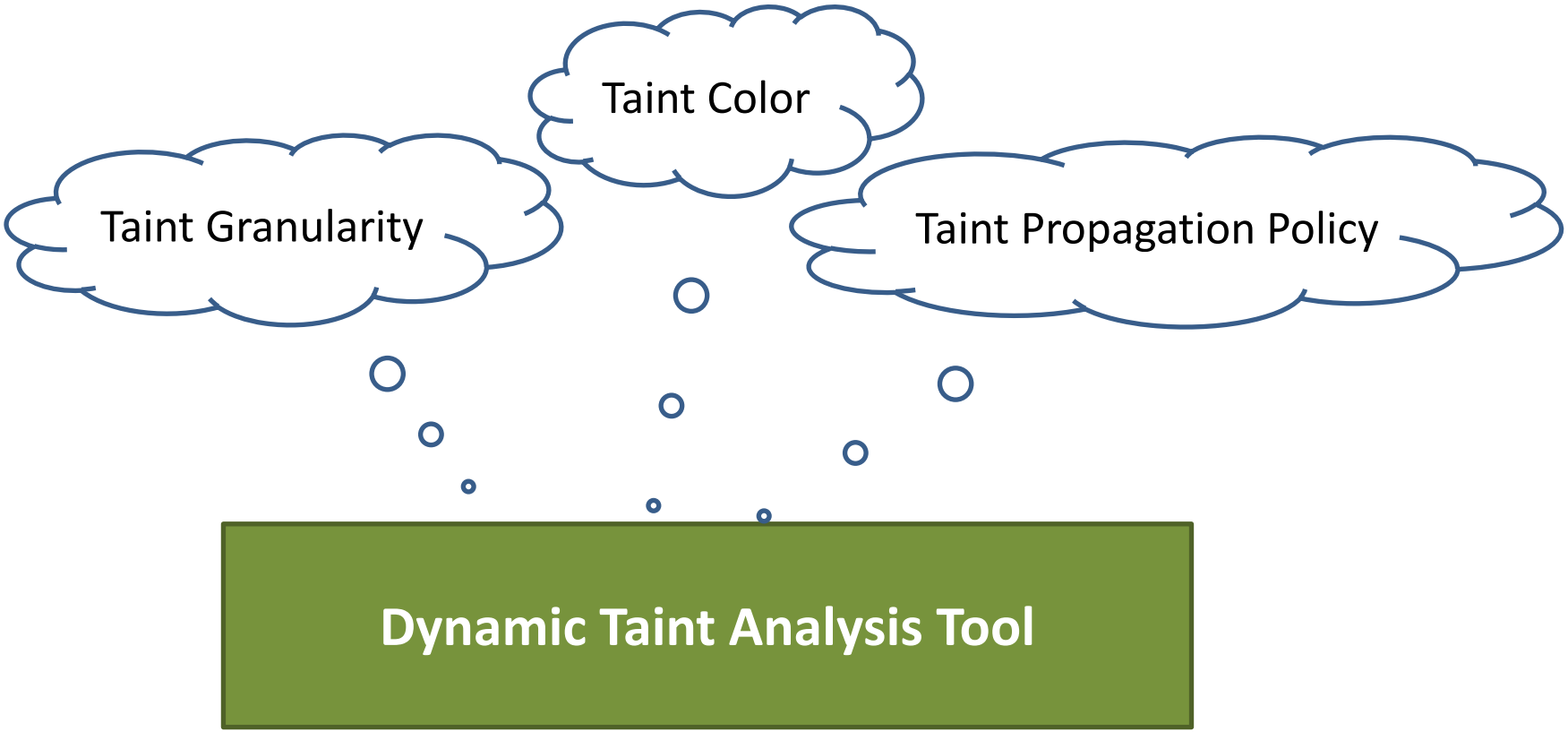
Dynamic Explicit Taint Propagation Analysis

- [说明]在程序运行过程中，通过实时监控程序的污点数据在数据流中的传播检测其能否从污点源传播到污点槽，比如重写攻击
- [对象]二进制指令
- [技术]
  1. 为污点数据扩展一个**污点标记**(tainted tag/mark)的标签并将其存储在存储单元(内存、寄存器、缓存等)
  2. 根据指令类型和指令操作数设计相应的**传播逻辑**，并依据执行轨迹分析污点标记的传播过程

The Taint Analysis Solution



Factors Considered in DTA Design



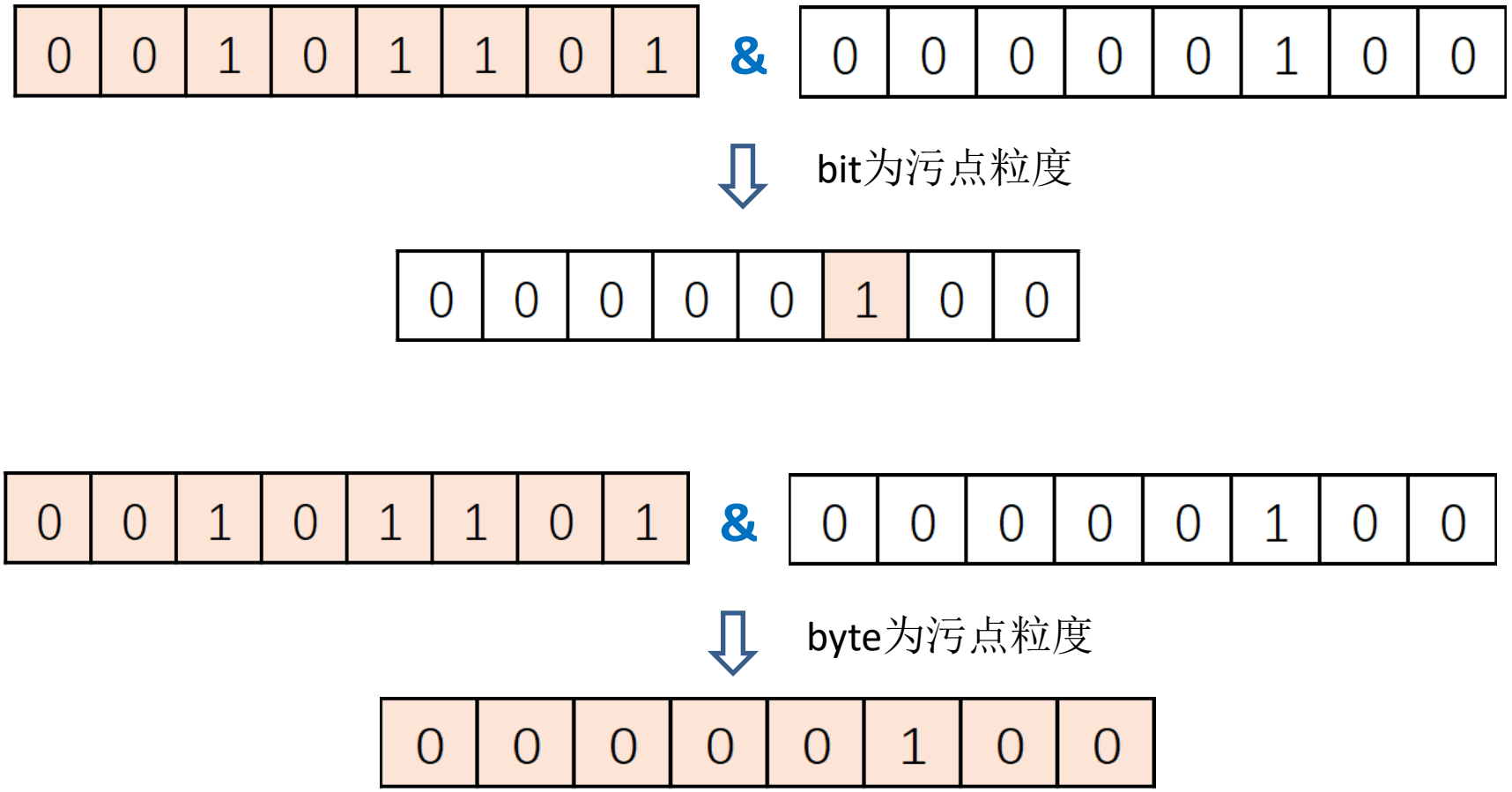
Factors Considered in DTA Design

- Taint Granularity
  - DTA 追踪污点的信息单位

    - 位粒度的DTA追踪寄存器/内存中的每一位是否被标记为污点
    - 字节粒度的DTA追踪每个字节是否被标记为污点
    - 字粒度的DTA追踪每个字是否被标记为污点

    ……
  - 直接影响DTA的准确性和性能
    - 字节粒度作为一种折中，被广泛使用

Taint Granularity Example



Factors Considered in DTA Design

- Taint Color
  - [含义]不仅想知道某个值是否被标记为污点，即其污染状态，而且想知道污点来自哪里
  - [解决方案]为每个污点使用不同的颜色
  - [所需成本]需要额外的开销存储污点颜色信息
  - [示例] 字节为污点粒度

Factors Considered in DTA Design

- 以**字节**为污点粒度的系统
  - 若只考虑字节是否被标记为污点，那么只额外需要**1位**信息，0 没有污染，1污染
  - 若需支持多污点颜色，那么需要多些开销？比如1字节，可以存储多少污点源？255？或其它？
  - 若为255则无法区别一个变量同时受多个不同污染源的影响
  - 一般只用**8种**颜色0x01, 0x02,0x04, 0x08, 0x10,0x20, 0x40, 0x80，通过“**按位或**”可以很容易区分污染源是哪个

Factors Considered in DTA Design

- Taint Propagation Policy
  - A propagation policy describes how taint markings should be propagated during execution.
  - Given a statement s, the taint markings for the data produced by s (produced data) are some function (mapping function) of the taint markings associated with the data that affects the outcome of s (affecting data)

    - produced data: identified unambiguously
    - mapping function
    - affecting data

    } different ways in which to identify affecting data and define the mapping function

Factors Considered in DTA Design

Taint Propagation Policy Example (byte with 2 colors)[1]

操作	指令	操作数污点（输入，4字节）		操作数污点（输出，4字节）	污点合并运算
		a	b	c	
c = a	mov	R B R B		R B R B	:=
c = a ⊕ b	xor	R R	B RB B RB	RB RB B RB	U
c = a + b	add	R R R	B B	R R B RB	U
c = a ⊕ a	xor	RB RB B RB			ϕ
c = a << 6	shl	R		R R	<<
c = a << b	shl		R	R R R R	:=

[1] 丹尼斯 安德里斯著，刘杰宏等译. 二进制分析实战. 人民邮电出版社，2021.

Taint Propagation Policy[1]

program	::=	*stmt**
stmt s	::=	var := exp | store(exp, exp) | goto exp | assert exp | if exp then goto exp else goto exp
exp e	::=	load(exp) | exp $\Diamond_b$ exp | $\Diamond_u$ exp | var | get_input(src) | v
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Table I: A simple intermediate language (SIMPIL).

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Figure 2: The meta-syntactic variables used in the execution context.

[1] E. J. Schwartz, T. Avgerinos and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2010, pp. 317-331, doi: 10.1109/SP.2010.26..

Taint Propagation Policy

taint t	::=	**T** | **F**
value	::=	$\langle v, t \rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Table II: Additional changes to SIMPIL to enable dynamic taint analysis.

Component	Policy Check
$P_{\textbf{input}}(\cdot)$, $P_{\textbf{bincheck}}(\cdot)$, $P_{\textbf{memcheck}}(\cdot)$	**T**
$P_{\textbf{const}}()$	**F**
$P_{\textbf{unop}}(t)$, $P_{\textbf{assign}}(t)$	t
$P_{\textbf{binop}}(t_1, t_2)$	$t_1 \vee t_2$
$P_{\textbf{mem}}(t_a, t_v)$	t_v
$P_{\textbf{condcheck}}(t_e, t_a)$	$\neg t_a$
$P_{\textbf{gotocheck}}(t_a)$	$\neg t_a$

Table III: A typical tainted jump target policy for detecting attacks. A dot $(\cdot)$ denotes an argument that is ignored. A taint status is converted to a boolean value in the natural way, e.g., **T** maps to true, and **F** maps to false.

Taint Propagation Policy

$$\frac{v \text{ is input from } src}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{get\_input}(src) \Downarrow \langle v, P_{\textbf{input}}(\text{src})\rangle}\ \text{T-INPUT} \qquad \frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash v \Downarrow \langle v, P_{\textbf{const}}()\rangle}\ \text{T-CONST}$$

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash var \Downarrow \langle \Delta[var], \tau_\Delta[var]\rangle}\ \text{T-VAR} \qquad \frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\textbf{mem}}(t, \tau_\mu[v])\rangle}\ \text{T-LOAD}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \Diamond_u e \Downarrow \langle \Diamond_u v, P_{\textbf{unop}}(t)\rangle}\ \text{T-UNOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\textbf{bincheck}}(t_1, t_2, v_1, v_2, \Diamond_b) = \mathbf{T}}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow \langle v_1 \Diamond_b v_2, P_{\textbf{binop}}(t_1, t_2)\rangle}\ \text{T-BINOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle \quad \Delta' = \Delta[var \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[var \leftarrow P_{\textbf{assign}}(t)] \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \tau_\mu, \tau'_\Delta, \Sigma, \mu, \Delta', pc+1, \iota}\ \text{T-ASSIGN}$$

$$\frac{\begin{array}{c}\iota = \Sigma[pc+1] \quad P_{\textbf{memcheck}}(t_1, t_2) = \mathbf{T} \\ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \tau'_\mu = \tau_\mu[v_1 \leftarrow P_{\textbf{mem}}(t_1, t_2)]\end{array}}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \tau'_\mu, \tau_\Delta, \Sigma, \mu', \Delta, pc+1, \iota}\ \text{T-STORE}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t\rangle \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc+1, \iota}\ \text{T-ASSERT}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_2\rangle \quad P_{\textbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-TCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 0, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\textbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_2]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_2, \iota}\ \text{T-FCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t\rangle \quad P_{\textbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-GOTO}$$

Figure 5: Modified operational semantics of SIMPIL that enforce a taint policy P. $\mathbf{T}$ denotes true.

The General Process

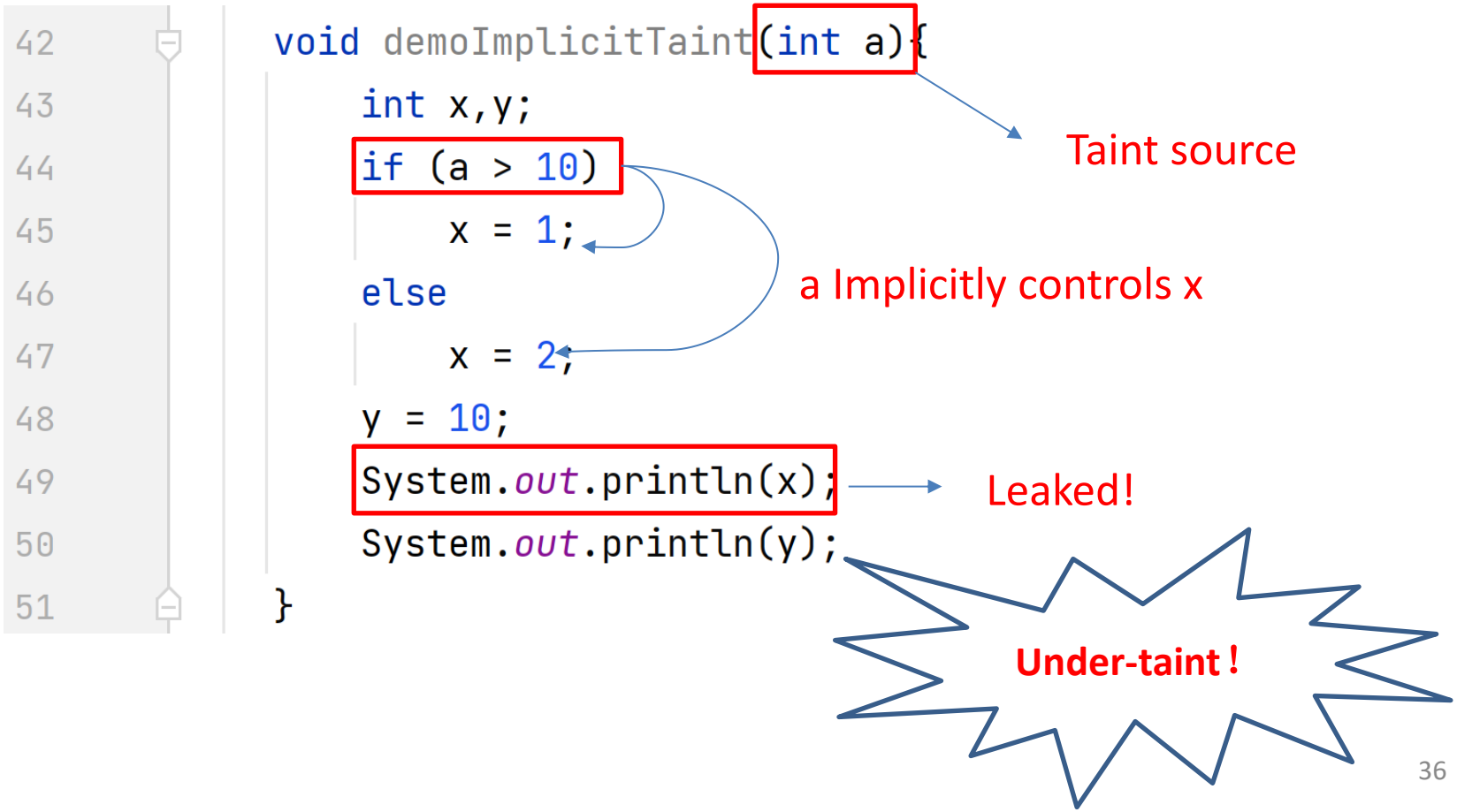
- Step 2: Taint Propagation Analysis
  - 4 Approaches
    1. Static Explicit Taint Propagation Analysis
    2. Dynamic Explicit Taint Propagation Analysis
    3. Static Implicit Taint Propagation Analysis
    4. Dynamic Implicit Taint Propagation Analysis

Data Flow based

Control Flow based

Implicit Taint Analysis Example

- **[隐式流分析]** 分析污点数据如何通过**控制依赖**进行传播



Implicit Taint Propagation Policy [1]

- Given $\mathbf{b_r}$ as a conditional branching statement that decides whether a statement $\mathbf{s_t}$ may be executed
  - the values that affect $\mathbf{b_r}$**'s** outcome may affect the value of the data modified by $\mathbf{s_t}$**,** the taint markings associated with b_r's source operands must be combined and associated with s_t's destination operands

[1] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07.

An Implicit Taint Analysis Approach

- Postdominator Tree[1]
  - Given two nodes **m** and **n** in a **CFG**, **n postdominates m**, denoted as n pdom m or pdom(m) = n **iff**

    all directed paths from **m** to **exit** contain **n**
  - Given two nodes **m** and **n** in a **CFG**, **n immediately postdominates m**, denoted as n ipdom m or **ipdom(m) = n iff** **n pdom m** and there is **no node o** such that **n pdom o** and **o pdom m**.

[1] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07.

An Implicit Taint Analysis Approach

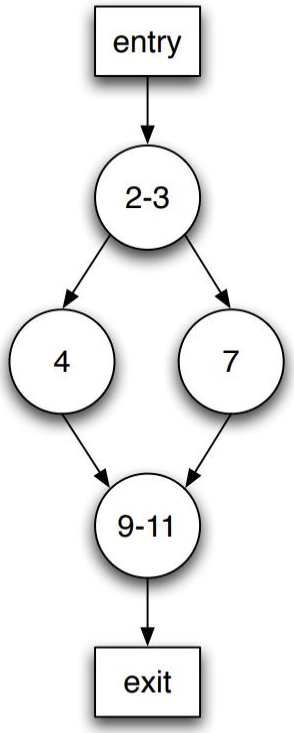
- Postdominator Tree (Cont.)
  - A **pdom tree** for a CFG is a **rooted tree** such that
    1. it has the same set of nodes as the CFG
    2. its root is the exit node
    3. each node **immediately postdominates** its direct descendents in the tree.
  - Given two nodes **m** and **n** in a CFG, **n** is control dependent on **m iff**
    1. There is a path P from m to n with any node o in P (excluding m and n) postdominated by n
    2. m is not postdominated by n.

An Implicit Taint Analysis Approach

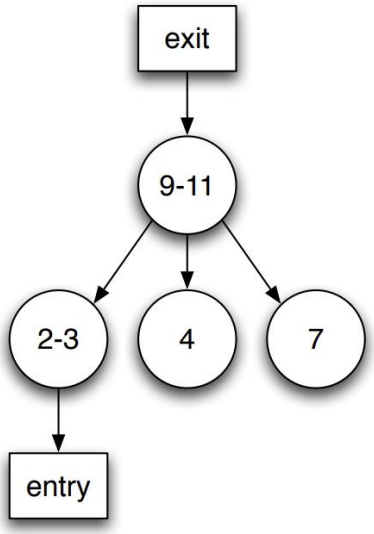
- **Control flow Based Taint Propagation**
  - When the execution reaches a conditional branching statement br, (1) computes a set taint that contains the combination of the taint markings for br's source operands and (2) add to a set S a pair < br, taint >.
  - When the execution reaches ipdom(br), where br is a conditional branching statements, it removes from S all pairs < x, y > such that x is equal to br.
  - When a statement st is executed and S is not empty, it adds, for each pair < x, y > in S, the taint markings in y to the set of taint markings to be combined and associated to st's destination operands

An Implicit Taint Analysis Approach

```
1   void demoImplicitTaint(int a){
2       int x,y;
3       if (a > 10){
4           x = 1;
5       }
6       else{
7           x = 2;
8       }
9       y = 10;
10      System.out.println(x);
11      System.out.println(y);
12  }
```



CFG



Pdom Tree

执行demoImplicitTaint(20)的污点传播过程：

Init: S = ϕ → Line3: S = { <3, {ta}>} → line4: S = {<3, {ta}>} ,<x, {ta}>} → line9: ipdom(3) S = {<x,{ta}>}

Over-taint

- 污点标记的数量过多而导致污点变量大量扩散的问题称为过污染问题(over-taint)
- 如何选择合适的污点标记分支进行污点传播

```
57  void demoOverTaint(){
58      int a = source();
59      int x = 0, y = 0, z = 0;
60      int w = a + 10;
61      if (a == 10){
62          x = 1;
63      }else if(a > 10 && w <= 23){
64          y = 2;
65      }else if(a < 10){
66          z = 3;
67      }
68      sink(x,y,z);
69  }
```

泄露的信息范围不同
1）a == 10中a的值，根据sink-x一次即可猜出
2）a > 10 && a <= 13
根据sink-y最多三次可猜出11，12，13
3）a < 10根据sink-z的值猜出的概率小，可以考虑不作为污点传播分支

Big Picture

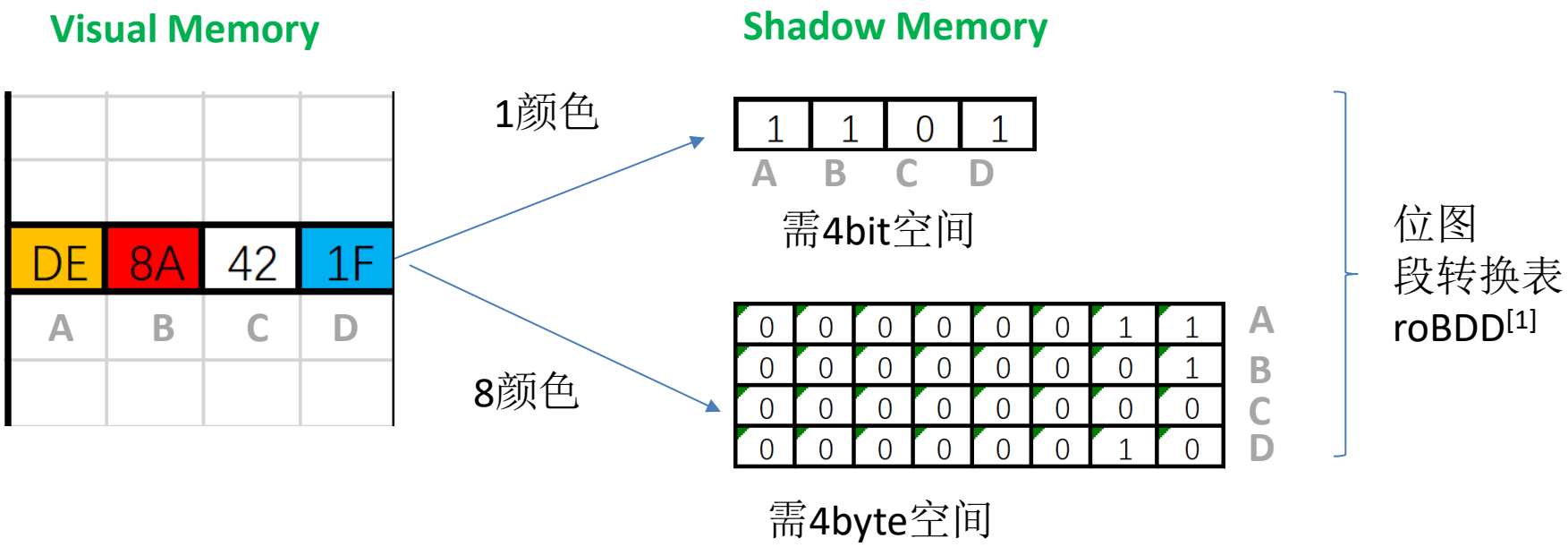
- Key Concepts
- Taint Analysis Approaches
- Dynamic Taint Analysis Framework
  - Libdft[1]
  - Dytan[2]
- Typical DTA Applications

[1] https://www.cs.columbia.edu/~vpk/research/libdft/

[2] https://github.com/behzad-a/Dytan

Shadow Memory

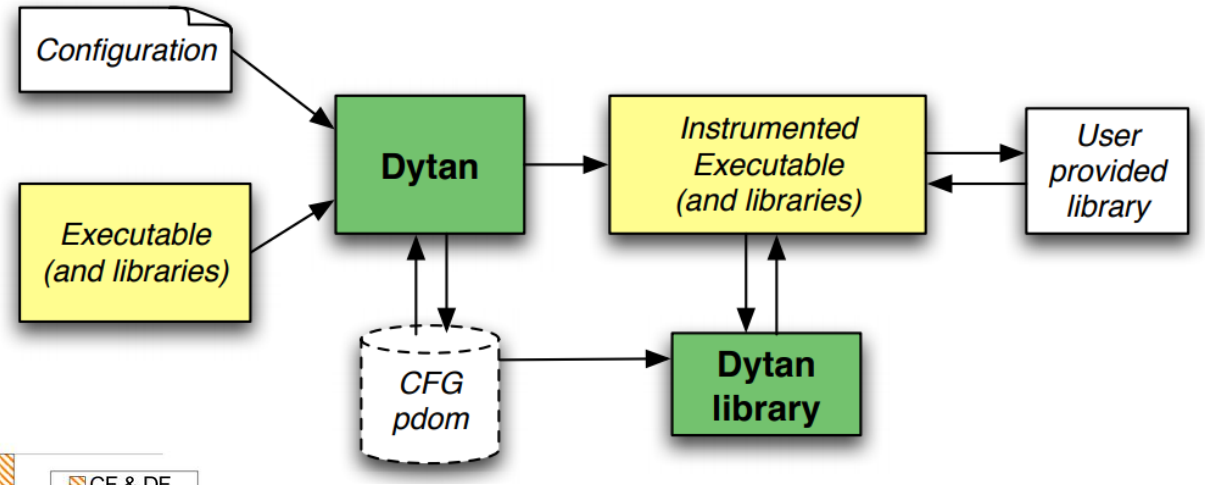
- The memory area that is used to store taint tags which is maintained by a DTA tool
- There is another special memory area used to store taint tags of CPU Registers only

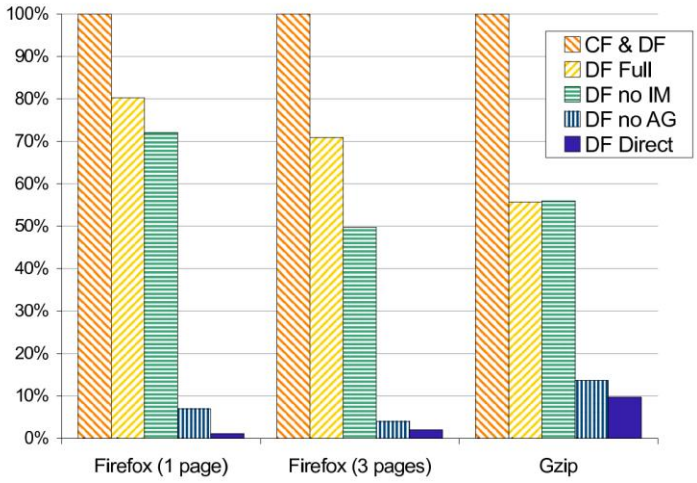


[1] 王铁磊 面向二进制程序的漏洞挖掘关键技术研究[D]. 北京大学，2011

The DyTAN Framework[1]

```
<dytan-config>
<sources>
  <source type=''network''>
    <host>*</host>
    <port>*</port>
  </source>
</sources>
<propagation>
  <dataflow>true</dataflow>
  <controlflow>false</controlflow>
</propagation>
<sinks>
  <sink>
    <id>36</id>
    <location type=''instruction''>
      <instruction=''ret''/>
      ...
      <instruction=''jmp''/>
    </location>
    <action=''validate-absence''/>
  </sink>
</sinks>
</dytan-config>
```





[1] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07).

Big Picture

- Key Concepts
- Taint Analysis Approaches
- Dynamic Taint Analysis Framework
- Typical DTA Applications
  - Protocol Reverse
  - Fuzz Testing

Protocol Reverse

[1] Jiayi Jiang, Xiyuan Zhang, Chengcheng Wan, Haoyi Chen, Haiying Sun and Ting Su. BinPRE: Enhancing Field Inference in Binary Analysis Based Protocol Reverse Engineering. In Procs of ACM CCS 2024. Salt Lake City, U.S.A., 2024.

- Code released on github (https://github.com/ecnusse/BinPRE.git)

Protocol Reverse

[2] Yuyao Huang, Hui Shu, Fei Kang, and Yan Guang. 2022. Protocol Reverse-Engineering Methods and Tools: A Survey. Comput. Commun. 182, C (Jan 2022), 238–254.

[3] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07).

[4] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In Proceedings of the 4th International Conference on Information Systems Security (ICISS '08). Springer-Verlag, Berlin, Heidelberg, 1–25.

Fuzz Testing

[1] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2011. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution. ACM Trans. Inf. Syst. Secur. 14, 2, Article 15 (September 2011), 28 pages.

[2] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, USA, 474–484.

[3] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: data flow sensitive fuzzing. In Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20). USENIX Association, USA, Article 145, 2577–2594.

Summary

- Taint Analysis is an application of Information Flow Analysis by tracking the use of tainted data
- Taint analysis approaches can be classified into two categories: static and dynamic, according to whether executing the program or not
- There are three steps in a taint analysis process: identify taint source and sink, taint propagation analysis and taint delete
- Taint propagation policies specify how the taint mark should be calculated when executing a certain instruction. Different affecting data and mapping function decide different policy
- Under-taint and over-taint problems are challenges of taint analysis

The End